An Automatic Translation Scheme from Prolog to the Andorra Kernel Language

Francisco Bueno
bueno@fi.upm.es

Manuel Hermenegildo †
herme@fi.upm.es *or* herme@cs.utexas.edu

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain

Abstract

The Andorra family of languages (which includes the Andorra Kernel Language –AKL) is aimed, in principle, at simultaneously supporting the programming styles of Prolog and committed choice languages. On the other hand, AKL requires a somewhat detailed specification of control by the user. This could be avoided by programming in Prolog to run on AKL. However, Prolog programs cannot be executed directly on AKL. This is due to a number of factors, from more or less trivial syntactic differences to more involved issues such as the treatment of cut and making the exploitation of certain types of parallelism possible. This paper provides basic guidelines for constructing an automatic compiler of Prolog programs into AKL, which can bridge those differences. In addition to supporting Prolog, our style of translation achieves independent and-parallel execution where possible, which is relevant since this type of parallel execution preserves, through the translation, the user-perceived "complexity" of the original Prolog program.

1 Introduction

A desirable goal in logic programming language design is to support both the don't-know nondeterministic, search-oriented programming style of Prolog and the don't-care indeterministic, concurrent communicating agents programming style of committed-choice languages. Furthermore, from an implementation point of view it is interesting to be able to support the or- and independent and-parallelism often exploited in the former (e.g. [Lus88, AK90, Kal87, HG90]) as well as the dependent and-parallelism exploited in the latter (e.g. [Cra90, IMT87, HS86]). The Andorra family of languages is aimed at simultaneously supporting these two programming paradigms and their associated modes of parallel execution. The Andorra proposal in [War88] (called the "basic" andorra model, on which the Andorra-I system [SCWY90] is based) defined a framework which allowed or-parallelism and also the and-parallel execution of *determinate* goals (determinate "stream and-parallelism"), this now being called the "Andorra Principle."

An important idea behind the choice of control in the basic Andorra model is to perform the least possible amount of computation while allowing the maximum amount of parallelism to be exploited. Another and complementary way of achieving this goal which has also been identified

[HR89, HR90] is to also run in parallel *nondeterminate* goals, but provided (or while) they are independent ("independent and-parallelism" –IAP). In order to also include this type of parallelism the *Extended* Andorra Model (EAM) [War90, HJ90] defines an execution framework which allows IAP in addition to the forms of parallelism supported in the basic Andorra model. The EAM defines rules which specify a series of admissible steps of computation from each possible given state. Several rules can be admissible from a given state and this gives rise to both nondeterminism and indeterminism, and also to opportunities for parallel execution. One important issue within this framework is thus that of control: i.e. which of the admissible rules should be applied in order to achieve the most efficient execution while attaining the maximum parallelism.

Two obvious approaches to treating the above mentioned issue are to put control decisions in the hands of the programmer or to try to do this automatically by compile-time and/or run-time analysis. The Andorra Kernel Language (AKL) [HJ90, JH91], uses explicit control. In particular, AKL allows (dependent) parallel execution of determinate subgoals, as stated by the Andorra Principle, but it also allows the more general forms of parallel execution of the EAM, albeit controlled by the programmer. The specification of control is done, among other mechanisms, by positioning the goals and constraints before or after a *guard operator*, in a way that can be reminiscent of the labeling of unification as input or output (i.e. ask or tell constraints [Sar89]) in the GHC language [Ued87a]. These operators divide body clauses into two parts, the guard and the actual body. Guards are executed in independent environments and proceed unless they attempt to perform output unification, while bodies wait until guards are completely solved and goals in the body promoted. Such goals are then executed concurrently provided they are determinate, in the spirit of the Andorra Principle. These properties give a means of control to the programmer which can be used to achieve parallel execution of general goals.

The AKL is therefore quite a powerful language. However, it does put quite a burden on the programmer in requiring certain specification of control. In particular, Prolog programs cannot always be executed directly on the AKL. This is due to a number of factors, from more or less trivial syntactic differences to more involved issues such as the treatment of cut, labeling of unification, and making the exploitation of certain types of parallelism, most notably IAP, possible without user involvement and preserving the programmer-perceived complexity of the original program.

The objective of this paper is to investigate how the above mentioned differences can be bridged, through program analysis and transformation. It points out the non-trivial problems involved in performing such a translation, and then provides solutions for these problems. Although desirable, our aim at this point is not to provide the best possible translation, which would take advantage of AKL properties to achieve a large reduction of search space, but rather to bridge the gap between Prolog and AKL in a manner that no increment in the search space is done, and also IAP can be exploited (with the important result of achieving "stability" in the frame of AKL for these cases). Building on partial translation approaches presented in [JH90, Her90] the paper presents a basic algorithm for constructing a translator from Prolog to AKL[1]. An important feature of the translation approach proposed herein is that it automatically detects and allows the parallel execution of independent goals (as well of course as or-parallelism, and the parallel execution of determinate goals even if they are dependent as per the Andorra Principle). The execution of independent goals in parallel has the very desirable properties of preserving the program complexity perceived by the programmer [HR89]. Important requirements for such a translation are the compile-time detection of goal independence and input/output modes. This requires in general a global analysis of the program, perhaps using abstract interpretation. In the approach proposed herein heavy use will be made of our compile-time tools, developed in the context of &-Prolog [HG90]. In particular, Prolog programs are first analyzed and annotated as &-Prolog programs (thus making goal independence

[1] Ueda [Ued87b] proposed automatic translation from Prolog to a committed-choice language (GHC, in his case). However, our aim and target language are quite different.

explicit), and then they are translated into AKL.

In the following section, the AKL control model and its rules are briefly reviewed together with some syntactic conventions. Then transformations for Prolog constructions for a basic translation are presented in section 3 and some rules for combining the AKL model with our purpose of achievement of independent parallelism are shown in section 4. Section 5 will present the analysis tools and why they are needed in the translation process. In section 6 some results are shown for the execution of a number of benchmarks automatically translated, and section 7 presents some conclusions.

2 The Andorra Kernel Language Revisited

In this section we present a brief overview of the AKL model of execution, in order to make the paper self-contained. The purpose is to, based on an understanding of this, extract the correct rules for a translation of Prolog which achieves the desired results. AKL and its model of execution have been fully described in [JH91, HJ90].

AKL is a language with *deep* guards. Thus, clauses are divided into two parts: the *guard* and the *body*, separated by a *guard operator*. Guard operators are: *wait* (:), *cut* (!), and *commit* (|). The following syntactical restrictions apply:

- Each clause is expected to have one and only one guard operator;
- All clauses in the definition of a predicate have to be guarded by the same guard operator. So, if any of the clauses is not guarded, the guard operator of its companions is assumed and positioned just after the clause neck.
- A wait operator is assumed, and in the above mentioned position, where no other operator can be assumed using the above mentioned rules.

Guards are regarded as part of clause selection. This means that a clause body is not entered unless head unification succeeds and its guard is completely solved. Then, execution proceeds by "expansion" of the present configuration by application of a rule of the computation model. The rules in the AKL model allow rewriting of configurations (states) leading to valid configurations from valid ones. They are fully described in [JH91], so we will simply enumerate them, providing *very informally* the concept behind the rule, rather than a precise definition:

1. *Local forking:* unfolds an atomic goal into a *choice* of all the alternatives in its definition (but without creating "copies" [2] yet of continuation goals).
2. *Nondeterminate promotion:* promotes *one* guarded goal with solved guard in a choice of several of them (i.e. copies the goal to the parent continuation, applying its constraint/substitution to it, and creates a "copy" of the continuation environment).
3. *Determinate promotion:* special case of the above when there is a *single* guarded goal in a choice if its guard is solved (no copying of the continuation environment is necessary).
4. *Failure and synchronization rules:* remove or fail configurations in the usual way.
5. *Pruning rules:* handle the effects of pruning guard operators.
6. *Distribution and bagof rules:* do the distribution of guards and the bagof operation.

[2] Although we refer to "copying" throughout the paper, part of the continuation goals could in principle be shared [War90].

These rules basically represent the allowable transitions of the EAM. The last three rules are less relevant for our purposes. In addition to these rules there are three basic control restrictions in the general computation model (meta-rules) which control the application of the above rules and which are highly relevant to our independent style translation:

- Pruning in AKL has to be *quiet*, that is, a solution for the guard of a cut or commit guarded clause may not further restrict (or constrain) variables outside its own configuration.
- Goals in the guard of a clause are completely and *locally* executed. This means that execution of guards is simultaneous but independent of the parent environment.
- Nondeterminate promotion is only admissible within a *stable subgoal* of a configuration. A goal is stable if no rule is applicable to any subgoal, and no possible changes in its environment will lead to a situation in which a rule is applicable in the goal.

As we shall soon see these three restrictions force the conditions under which translation has to be done if we want to achieve parallelism and correct pruning in the translated clauses. But first, we will illustrate the AKL execution model with a simple example:

```
partition([],_,Left,Right):- |,
        Left = [],
        Right = [].

partition([E|R],C,Left,Right):-
        E < C, |,
        Left = [E|Left1],
        partition(R,C,Left1,Right).
partition([E|R],C,Left,Right):-
        E >= C, |,
        Right = [E|Right1],
        partition(R,C,Left,Right1).
```

For a query such as `partition([2,1],3,I,D)` the initial configuration would be a choice-point with the three clauses for the predicate. Head unification would fail the first alternative (`[]=[2,1]`), but the second one would succeed (`[E|R]=[2,1],C=3,E<C`, including the guard), thus pruning the rest of the alternatives.

The single remaining alternative would then be promoted by determinate promotion, its bindings published, and execution would proceed with goals in its body. Note that this could not be done if, for example, the goal `Left=[E|Left1]` were included in the guard, as it would add constraints to the variable `I` (`I=Left`) in the external configuration, and thus pruning would have been "noisy."

On the other hand, if the clauses had no (explicit) guard operator, a wait operator would be assumed. In this case, both the second and third alternatives would succeed and only nondeterminate promotion would be applicable. If the configuration is stable, and assuming that the rightmost alternative is selected for promotion, the goal `E>=C` (i.e. `2>=3`) would be executed (and failed) *only* after promotion. After failure of this branch, determinate promotion of the remaining one would be applicable, and execution would proceed as before.

3 Translating Prolog Constructions

Having the aforementioned rules in mind, we now discuss transformation rules for translating basic Prolog constructions, disregarding any possible exploitation of IAP. Even this straightforward step is nontrivial, as we shall soon see. This is due mainly to the semantics of cut in both Prolog and

AKL, cut being a guard operator in the latter. With the restrictions required for guard operators to achieve both syntactic and semantic correctness in AKL, we find problems in the following constructions:

- syntactical restrictions:
  - definitions of predicates in which a pruning clause appears,
  - clauses in which more than one cut appears;
- semantic restrictions:
  - if-then-elses, where the cut has a "local" pruning effect,
  - pruning clauses where the cut is regarded as *noisy* (i.e. attempts to further restrict variables outside its scope),
  - side-effects and meta-logical predicates, which should be sequentialized.

The transformations required to deal with these constructions are proposed in the following subsections. This is done mainly through examples. The aim is thus not to provide precise and formal definitions of program transformations but rather to provide the intuition behind the process of translation. In subsequent sections we will discuss other issues involved in the process of translation, such as achievement of IAP, problems in this, and its relation with the AKL stability conditions.

3.1 Direct translation

First, as all AKL clauses in a definition are forced to have the same guard operator, we have to ensure this is achieved. For example:

Example 1 Same guard operator in a definition

```
p(X,Y):- q(X), r(Y).                    p(X,Y):- q(X), r(Y).
p(X,Y):- test(X), !, output(Y).         p(X,Y):- pc(X,Y).
p(X,Y):- s(X,Y).
                                        pc(X,Y):- test(X), !, output(Y).
                                        pc(X,Y):- s(X,Y).
```

Note that clauses before the pruning one will have an (assumed) wait operator and clauses after that one (and that one itself) will have an (assumed) cut operator. All of them but the pruning one have an empty guard. Note that, had the program not been rewritten, the rules for assuming guard operators would have put a cut operator in the first clause, which is obviously not the correct translation.

Note also, that only one guard operator is to be allowed in a clause. Therefore repeated cuts in the same body (which are otherwise strongly discouraged as a matter of style and declarativeness) have to be "folded" out using the technique sketched below:

Example 2 Single guard operator in a clause

```
p(X,Y):-  test(X), !,                   p(X,Y):- test(X), !, foo(X,Y).
          test(Y), !,
          accept(X,Y).                  foo(X,Y):- test(Y), !, accept(X,Y).
```

Second, the AKL cut operator is regarded as a *guard* operator, and, furthermore, it has to be quiet (which is not the case in some Prolog constructions, which cannot be easily translated to AKL). One of them is local pruning, i.e. if-then-else. Indeed, an if-then-else can be viewed as a disjunction containing a cut whose scope is limited to the disjunction itself, rather than the clause in which it appears. Thus the following preprocessing can be done:

Example 3 Local pruning of if-then-else

```
p(X):- ( cond(X) -> q(X,Y)                p(X):- foo(X,Y,Z), s(Y,Z).
                  ; r(X,Z)
       ),                                 foo(X,Y,_):- cond(X), !, q(X,Y).
       s(Y,Z).                            foo(X,_,Z):- r(X,Z).
```

Last but not least, we have to ensure the quietness of all AKL cuts. A cut is quiet if it does not attempt to bind variables which are seen from outside its own scope, that is, the clause where they appear. Then, if this is not the case, we have to make that binding explicit in the form of an equality constraint (a unification) and place it after the cut itself, i.e. outside the guarded part of the clause:

Example 4 Making a cut quiet

```
p(X,Y):- test(X), output(Y), !.          p(X,Y):- test(X), output(Y1), !, Y1=Y.
p(X,Y):- s(X,Y).                         p(X,Y):- s(X,Y).
```

Note that knowledge of input/output modes of variables is required for performing this transformation, and that the transformation may not always be safe[3]. This will be discussed in the following subsection.

3.2 Noisiness of cut

The main difference between cut in Prolog and cut in AKL is that cut is *quiet* in AKL[4]. "Quiet" in the context of a cut means that the solution of the cut's guard is quiet, that is, it does not add constraints to variables outside the guarded goals themselves, other than those which already appear in its environment.

Indeed, a transformation such as the one proposed in example (3.1).4 can make a noisy cut quiet. What it does is to delay output unification until the guard is promoted by making it explicit in the body part of the clause. We regard a variable to be *output* in a query if execution for this query will further constrain it; a variable will be regarded as *input* if execution will depend on its state of instantiation (or constraint). In other words, a variable is an output variable in a literal if it is further instantiated by the query this literal represents, it is an input variable if it makes a difference for the execution of the literal whether the variable is instantiated or not[5]. Note that a given variable can be both input and output, or none of them.

The objective of a transformation such as the one proposed is to rename apart all output variables in the head of a pruning clause, and then bind the new variables to the original ones in the body of the clause, leaving input variables untouched. In general, it is unwise to rename apart input variables since, from their own definition, this renaming would make the variable appear uninstantiated and potentially result in growth in the search space of the goals involved. This would not meet our objective of preserving the complexity of the program (and perhaps not even that of preserving its semantics). However, since a variable can be both input and output a conflict

[3] Note also that this transformation, when safe, may be of advantage as well in standard Prolog compilers in order to avoid trailing overhead.

[4] Nevertheless, a *noisy* cut has also been implemented in AKL, which we will discuss later.

[5] These definitions are similar to those independently proposed in [SCWY91], (and also in the spirit of those of Gregory [Gre85]), which describes translation techniques from Prolog to Andorra-I, an implementation of the Basic Andorra Model. Although the techniques used in such a translation have some relationship with those involved in Prolog-AKL translation, the latter requires in practice quite different techniques due to AKL being based on the *Extended* Andorra Model (thus having to deal with the possibility of parallelism among non-determinate goals and the stability rules) and the rather different way in which the control of the execution model (explicit in AKL and implicit in Andorra-I) is done in each language.

between renaming and not-renaming requirements appears in such cases. For these cases in which a variable cannot be "moved" after the cut guard operator a real noisy cut is needed. This operator exists in AKL (!!), together with a sequentialization operator, the sequential conjunction (&). It is necessary that every noisy cut be sequentialized, this to ensure that pruning would occur in the same context that it would in Prolog. Thus, every literal call to the pruning predicate has to be sequentialized to its right, and every other call to a predicate sequentialized has in turn to be also sequentialized. For this reason noisy cut is not very efficient, and thus the translation tries to minimize its use.

At this point we can summarize the action that should be taken in every case to transform the pruning clauses of a Prolog program, based on the knowledge of input/output variables, that is, whether they are "tested" or not and further instantiated or not. Here we use "noisy" to mean the transformation that defaults to the AKL noisy cut, and "move" to refer to the renaming of variables like in example (3.1).4.

Further Instantiated?	Tested?	Action
yes	yes	noisy
	no	move
	unknown	user
no	*	none
unknown	yes	user
	no	move
	unknown	user

Note that the knowledge of input/output modes in the Prolog program that is assumed in this transformation requires in general a global analysis of the program and can only be approximated, the translator having to make conservative approximations or warn the user ("user" cases above) when insufficient information is available. Note also that the "user" cases can be replaced by "noisy" cases if a non-interactive transformation is preferred. This subject will be discussed further in section 5, as well as the type of analysis required.

3.3 Synchronization of side-effects

In general, the purpose of side-effect synchronization is to prevent a side effect from being executed before other preceding (in the sense of the sequential operational semantics) side-effects or goals, in the cases when such adherence to the sequential order is desired. In our context, if side-effects are allowed within *parallel* AKL code and a behaviour of the program identical to that observable on a sequential Prolog implementation is to be preserved, then some type of synchronization code should be added to the program. In general, in order to preserve the sequential observable behaviour, side-effects can only be executed when every subgoal to their left has been executed, i.e. when they are "leftmost" in the execution tree. However, a distinction can be made between *soft* and *hard* side-effects (a side-effect is regarded to be *hard* if it could affect subsequent execution), see [DeG87] and [MH89]. This distinction allows more parallelism. It is also convenient in this context to distinguish between side-effect built-ins and side-effect procedures, i.e. those procedures that have side-effects in their clauses or call other side-effect procedures.

To achieve side-effect synchronization, various compile-time methods are possible:

- To use a chain of variables to pass a "leftmost token", taking advantage of the suspension properties of guards to suspend execution until arrival of the token [SCWY91].

- To use chains of variables as semaphores with some compact primitives that test their value. In [MH89] a solution was proposed along such lines, and its implementation discussed.

- To use a sequentialization built-in to make the side-effect and the code surrounding it wait; this primitive would be in our case the sequentialization operator "&".

In the first solution, a pair of arguments is added to the heads of relevant predicates for synchronization. Side-effects are encapsulated in clauses with a wait (:) guard containing an "ask" unification of the first argument with some known value (*token*), to be passed by the preceding side-effect upon its completion. Upon successful execution of the current side-effect the second argument is bound ("tell") to the known value and the token thus passed along. This quite elegant solution can be optimized in several cases.

The second solution can be viewed as an efficient implementation of the first one, which allows further optimization [MH89]. The logical variables which are passed to procedures in the extra arguments behave as semaphores, and synchronization primitives operate on the semaphore values.

In the third solution, every soft side-effect is synchronized to its left with the sequentialization operator "&", and every hard one both to its left and right. This sequentialization is propagated upwards to the level needed to preserve correctness. This introduces some unnecessary restrictions to the parallelism available. However, if side-effects appear close to the top of the execution tree, this may be quite a good solution.

4 Stability and Achievement of Independent And-Parallelism

In order to achieve more parallelism than that available by the translations described so far one might think of translating Prolog into AKL so that every subgoal could run in parallel unrestricted. However, this can be very inefficient and would violate the premise of preserving the results and complexity of the computation expected by the user. On the other hand, and as mentioned before, parallel execution of *independent* goals, even if they are nondeterminate, is an efficient and desirable form of parallelism and its addition motivated the development of the EAM, on which the AKL is based. Nevertheless, in AKL goals known to be independent have to be explicitly rewritten in order to make sure that they will be run in parallel. This is because of the rules that govern the (nondeterminate) promotion, that is, the stability condition on nondeterminate promotion, which will prevent these goals for being promoted if they try to bind external variables for output. Therefore, one important issue is the transformation that is needed to avoid suspension of independent goals. This is presented in section 4.1. Also, independence detection can and will be used to reduce stability checking, a potentially expensive operation.

Clearly, an important issue in this context is how stability/goal independence is detected. In the framework of the &-Prolog system we have already developed technology and the associated tools for determining independence conditions for goals and partially evaluating many of those conditions at compile-time through program analysis. Conceptual models for independent and-parallel execution have been presented and their correctness and efficiency proved [HR89]; among all we focus on the and-parallelism models proposed in [HR90, HR89]. For different but related models the reader is referred to the references in those papers. As mentioned before, in the translation process we propose to use algorithms and tools already developed in the context of &-Prolog. In this context, a series of algorithms used in the &-Prolog compiler for annotating Prolog programs have been implemented and described in [MH90]. These algorithms select goals for parallel execution and, using the sufficient rules proposed in [HR89], generate the conditions under which independence is achieved and therefore independent parallel execution ensured. The result is a transformation of a given Prolog clause into an &-Prolog clause containing parallel expressions which achieve such independent and-parallelism.

The output of this analysis is made available for the translation process in the form of an annotated &-Prolog program [HG90], i.e. the program itself expresses which goals are independent

and under which conditions. These conditions are expressed in the form of if-then-elses which have the intuitive meaning of "if the conditions hold then run in parallel otherwise sequentially." The parallelism itself is made explicit by using the "&" operator to denote parallel conjunction instead of the standard sequential conjunction denoted by ","[6]. Some new issues are involved in the interaction between the conditions of these parallel expressions and other goals run in parallel concurrently, as it would be the case in AKL. These will be presented in section 4.2.

4.1 The transformation proposed

At this point the &-Prolog conditionals are regarded as input to the translator. As such, if-then-elses are preprocessed in the form mentioned in the previous sections and the remaining issue is the treatment of the parallelization operator "&". In implementing this operator we will use the AKL property that allows local and unrestricted execution of guards, i.e., goals that are encapsulated in a guard can run in parallel with goals in other guards even if they are nondeterminate. The transformation that takes advantage of this will:

- put goals known to be independent in (different) guards, and
- extract output arguments from the guards, binding them in the body part of the clauses,

the last step being required so that the execution of these goals is not suspended because of their attempting to perform output unification. With the guard encapsulation we ensure that those predicates will be executed simultaneously and independently. The following example illustrates the transformation involved:

Example 5 Encapsulation of independent subgoals

```
p(X):- (ground(X), indep(Y,Z) ->
          q(X,Y) & r(X,Z)
        ; q(X,Y) , r(X,Z)
       ),
       s(Y,Z).
```

```
p(X):- pp(X,Y,Z), s(Y,Z).

pp(X,Y,Z):- ground(X), indep(Y,Z), !,
            qp(X,Y), rp(X,Z).
pp(X,Y,Z):- q(X,Y), r(X,Z).

qp(X,Y):- q(X,Y1), :, Y=Y1.

rp(X,Z):- r(X,Z1), :, Z=Z1.
```

When the condition is met, both subgoals will be tried by the local fork rule, then *both guards* will be completely and locally solved, and then, as goals are independent on X (X is ground) and no output is produced in the guard, the nondeterminate promotion rule is always applicable and all solutions will be tried in the standard cartesian product way. Thus, parallel execution is ensured for those goals that are identified as independent.

On the other hand, when the condition fails (the goals being dependent) they appear together in a body with an empty guard. This means that the guard will be immediately solved, the clause body promoted, and subgoals tried simultaneously. Then the standard stability and promotion rules will apply.

It should be noted that, as in the case of cut, and in addition to detecting goal independence, to be able to perform this transformation it is necessary to have inferred mode information regarding the predicate clauses. In section 5 techniques used in order to infer this information will be reviewed.

[6] Note that in AKL these operators have just the opposite meaning!.

4.2 Cohabitation of dependent and independent and-parallelism and stability checks

When evaluating the conditions of parallel expressions at run-time within a parallel framework such as that of the AKL, they may not evaluate to the same value than during a Prolog execution. This is what we have termed in another context the *CGE-condition problem* [GSCYH91][7], and may result in a loss (or increase) of parallelism. To deal with these issues, different levels of restrictions can be placed on the translation:

- Disallow any parallel execution except for those goals found to be independent.
- Allow parallel execution only for goals not binding variables that appear in the conditions or CGE.
- Allow parallel execution outside a CGE but sequentialize before and after the conditional parallel expressions.
- Allow unrestricted parallel execution unrestricted, i.e. no sequentialization is to be done.

The first solution can be implemented by translating every conjunction as a *sequential* AKL conjunction, except those joining independent goals. This will lead to a type of execution where only goals known to be independent are run in parallel and which directly resembles that of &-Prolog [HG90]. The same search space as &-Prolog will be explored. Nondeterminate (and determinate) promotion will then be restricted to only independent and sequential goals. Thus, one very important advantage of this translation is that *no checks on stability ever need to be done*, as stability is ensured for sequential and independent execution. This is an important issue since stability checking is a potentially expensive operation (and very closely related to independence checking). Thus, in an *ideal* AKL implementation code translated as above, i.e. free of stability checks, should run with comparable efficiency to that of &-Prolog. On the other hand, the transformation loses determinate dependent and-parallelism and its desirable effect of co-routining, which could be useful in reducing search space [SCWY90].

The second solution attempts to preserve the environment in which the CGE evaluates while allowing coroutining of goals that don't affect CGE conditions and goals. Although interesting, this appears quite difficult to implement in practice as it requires very sophisticated compile-time analysis and will probably incur in run-time overheads for checking of the conditions placed in the program.

The third solution can be viewed as a relaxation of the first one to achieve some coroutining, or as an efficient (and feasible) way of partially implementing the second one. Goals before and after are allowed to execute in parallel using the Andorra Principle, but they are sequentialized just before and after a CGE. In this way CGEs evaluate in the same context as in Prolog and the same level of independent and-parallelism is achieved. This translation has the good characteristics regarding search space of the previous one. In addition, some reduction of search space due to coroutining will be achieved. However, stability checking, although reduced, cannot in general be eliminated altogether.

The fourth solution will allow every goal to run in parallel. The full EAM and AKL operational semantics (including stability) has to be preserved. The execution of goals which are unconditionally independent or depend only on groundness checks (conditionals in the parallel expressions are

[7] Note that some other problems mentioned in [GSCYH91] regarding the interaction between independent and dependent and-parallelism (in particular, the *determinate goal problem*) are less of an issue in the proposed translation to AKL because independent goals execute in their own environments, thanks to the dynamic scoping of AKL guards. In any case, the AKL implementation is assumed to cope with all types of goal activations possible within the EAM.

composed of ground/1 and indep/2 checks, as in the example of section 4.1) will be the same as in &-Prolog as eager execution of other goals cannot affect ground or empty checks [GSCYH91]. However, independence checks may fail where they wouldn't in Prolog (therefore losing this parallelism), but also succeed where they would fail in Prolog (therefore gaining this parallelism). Also, the number of parallel steps will always be the same or less as in Prolog (although different than in &-Prolog). This solution (as well as the first and second ones) appear as quite reasonable compromises and offer different tradeoffs. The current translation approach uses this fourth option, but the others should also be explored.

5 Inferring modes - Abstract Interpretation

We have mentioned in previous sections the need for inferring modes of clause variables (i.e. whether they are input or output variables) in Prolog programs. The main reason for this need is that we have to know which are the output variables in a clause in order to rename them apart and place corresponding bindings for them in the body part of the clause in both

- the pruning clauses (as shown in section 3.2), and
- the remade clauses for parallel execution (as shown in section 4.1 in example 5).

Much work has been done in global analysis of logic programs to infer run-time properties, and, in particular, modes, mostly using the technique of abstract interpretation [CC77]. A more sophisticated sort of variable binding analysis (comprising *groundness*, *aliasing*, and *freeness* information) is instrumental in the process of inferring the independence conditions for literals in a body. While not strictly needed, such an analysis is extremely useful as it allows the reduction of the number of conditions and therefore the improvement of performance by reducing run-time checking [WHD88, MH92] (these papers provide references to the important body of other work in this area). The standard global analyzer in the &-Prolog compiler, described in [MH92], infers groundness and variable sharing/aliasing. Since variable freeness is also needed for the AKL translator, this analyzer has been extended to use the algorithm described in [MH91] and infer variable freeness information.

It turns out that freeness information is very useful for many reasons [MH91]. In the translation process it is essential for determining input/output arguments. This we can show by simply expressing the information required for the table in section 3.2 in terms of information directly available from abstract interpretation. In order to do this, recall, as defined in section 3.2, that a program variable (or an argument) is output in a literal if the call to the corresponding predicate further instantiates this variable, and it is input in a literal if its state of instantiation is going to be checked in the execution of the call for that literal. With these definitions in mind the following table shows how the input or output character of variables can be decided in a good number of cases based on the information directly available from global analysis:

Before	After	Output?	Input?
ground	(ground)	no	*
free	free	no	*
	semi	yes	no
	ground	yes	no
$semi_1$	$semi_1$	no	*
	$semi_2$	yes	?
	ground	yes	?

From the table we identify cases in which it is clear that the variable is known not to be an input variable, without any further analysis (i.e. when the variable is free). Furthermore, we realize that

if a variable is known not to be an output variable then it doesn't need to be renamed apart and it is not necessary to determine whether it is an input variable or not ("*" cases). Reducing the cases where knowing if a variable is input is quite useful since inferring whether a variable binding is needed or not requires additional analysis ("?" cases). This analysis seeks to decide if a variable is crucial in clause selection or checking. Note that the analysis has to be extended for every child procedure of the one being analyzed.

Finally, we would like to also mention that combining mode/type analysis (such as the one used in [SCWY91] or [Jan90]) with the accurate tracking of sharing and freeness information of [MH91] could be very helpful in this process (improving the ability to more accurately resolve different degrees of partial instantiation such as the $semi_1/semi_2$ cases in the table above) and is part of our plans for future work.

6 Performance Timings

This section presents some results on the timing of a number of benchmarks in a prototype AKL system. The AKL versions of the programs obtained through automatic compile-time translation are compared with versions specifically written for AKL. Timings for the original Prolog versions are also included for comparison and also with the intention of identifying translation paradigms that help efficiency. With this aim in mind, the set of benchmarks has been chosen so that performance results are obtained for several different programming paradigms, and a number of different translation issues are taken into account. The results show that translation suffices in most cases, provided state-of-art analysis technology is used.

Timings[8] have been done for the Prolog program (compiled and interpreted), the AKL program obtained from automatic translation and the "hand-written-AKL" version. Execution until the first solution is obtained has been measured. Timings are an average of ten consecutive executions done after a first one (not timed) and are given in in milisseconds, rounded up to tens.

We briefly introduce the programming paradigms represented by each of the benchmarks used. *qsort* has been translated in two ways, one that "folds" pruning definitions, and another one that is able to "extend" the cut to all clauses; the latter showing an advantage w.r.t. the former. *sort* illustrates the advantage of being able to detect that some cuts are not noisy (as opposed to defaulting to noisy cut in every case). In fact, in this case the translated version is slightly faster than the hand-coded one.

For *money* we have used three different versions. In the first version of the program the problem is solved through extensive backtracking. In the second one the ordering of goals is improved at the Prolog level. In the third version the Prolog builtins are translated into AKL specific ones. As in *zebra* the difference with the "hand-written" version is in the use of the arithmetic predicates: addition is programmed in the hand-coded AKL version as illustrated by the *sum/3* predicate,

```
sum(X,Y,Z):- plus(X,Y,Z0), |, Z = Z0.
sum(X,Y,Z):- minus(Z,Y,X0), |, X = X0.
sum(X,Y,Z):- minus(Z,X,Y0), |, Y = Y0.
```

in which the coroutining effect provides a "constraint solving" behaviour.

Scanner is a program where AKL can take a large advantage from concurrent execution and the "determinate-first" principle, even without explicit control, and this is shown in the good performance of the translated program. On the other hand, in *triangle* and *knights* heavy use of special AKL features has been made, through hand-optimization.

[8] SICStus 1.8 and a sequential AKL 0.0 prototype system, made available by SICS, have been used.

	SICStus compiled	AKL translated	AKL written
qsort (1st)	30	750	290
qsort	30	290	290
mergesort	20	870	910
money (1st)	66,590	294,370	530
money	47,790	294,070	530
money (built)	47,790	187,920	530
zebra	8,550	10,380	1,980
triangle	3,140	152,230	11,020
knights	79,960	1,165,020	480
scanner	1,407,450	540	120

In *matrix*, *hanoi*, *query*, and *maps* (and also *qsort*), encapsulation of different programming paradigms has been tried. The results show that encapsulating independent goals which are determinate provides no improvement, but performance improves when they are nondeterminate. Performance also improves in the case of goals which act in producer/consumer fashion (*maps*). These results suggest that AKL control similar to that of hand-coded versions can be imposed automatically for paradigms other than independence of goals.

	SICStus compiled	AKL encapsulated	AKL translated
qsort	30	290	290
matrix	50	610	690
hanoi	10	70	310
query	70	370	1,600
maps	90	140	2,240

The automatic transformation achieves reasonably good results when compared to code specifically written for AKL, provided one takes into account that the starting point is a Prolog program with little specification of control, and it is being compared to an AKL program where control has been greatly optimized by the programmer. The examples where the largest differences show are those in which the control imposed by hand in the AKL program changes the complexity of the algorithm, generally through smart use of suspension (as in the *sum/3* predicate), something that the transformation can not yet do automatically. However, the results also show that it would obviously be desirable to extend the translation algorithms towards implementing some of the smart forms of control that can be provided by an AKL programmer.

When comparing with Prolog, both the interpreted and compiled Prolog figures should be considered, as the AKL system prototype used is somehow something in between a compiler and an interpreter. The results show that a variable performance improvement can be obtained whenever determinism is significant in the problem (this is quite spectacular in *scanner*). Also, the encapsulation transformation can help efficiency in some cases. In any case the figures are of course preliminary and a more exhaustive study should clearly be done after improvements in the translation prototype and the AKL system, and also when an actual parallel AKL system is available.

7 Conclusions

We have presented an algorithm for translating Prolog into AKL which in addition achieves independent and-parallel execution of appropriate goals. We have pointed out a series of non-trivial

problems associated with such a translation and proposed solutions for them based on existing global analysis technology. We have shown how to take advantage both of the AKL execution model (the Extended Andorra Model) and the independence analysis performed in the context of &-Prolog to produce a translation that allows the exploitation of all the forms of parallelism present in AKL (dependent-and, independent-and, and or-parallelism) while offering the user the familiar Prolog (or, in general, logic with minimal control) view (and debugging ease!). Most importantly, this is done while preserving or improving the user-perceived complexity of the program. The transformation is relevant even in the case of a sequential AKL implementation since the reduction of stability checking which follows from knowledge of goal independence can already be of significant advantage, given the expected cost of stability tests. In the case of a parallel AKL implementation the transformation amounts to a form of automatic parallelization and search space reducing implementation for Prolog programs which exploits the EAM, and imposes a particular form of control on it.

A *sequential* AKL implementation is already being developed at SICS with a first prototype already running. The translator itself is also being implemented and a preliminary version is already integrated with the &-Prolog system compilation tools. The combination has been tested and some sample programs executed successfully on AKL, and compared with their specific AKL counterparts. Further work is expected in the translator as better translation algorithms are developed to take more specific advantage of the AKL control facilities, in particular coroutining, in more accurately detecting input and output variables, in adapting the algorithms to possible evolutions of the AKL, in evaluating the performance of the translated programs with respect to Prolog, and in the formal proof of the correctness of the transformation and its preservation of user expected computation size, the latter point being supported already in part by the basic results on independent and-parallelism.

References

[AK90] K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, October 1990.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[Cra90] Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming*. MIT Press, 1990.

[DeG87] D. DeGroot. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming*, pages 80–89. San Francisco, IEEE Computer Society, August 1987.

[Gre85] S. Gregory. *Design, Application and Implementation of a Parallel Logic Programming Language*. PhD thesis, Imperial College of Science and Technology, London, England, 1985.

[GSCYH91] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: A Model Intergrating Dependent-, Independent-, and Or-parallelism. Technical report, University of Bristol, March 1991.

[Her90] M. Hermenegildo. Compile-time Analysis Requirements for the Extended Andorra Model. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, June 1990.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HS86] A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

[IMT87] N. Ichiyoshi, T. Miyazaki, and K. Taki. A Distributed Implementation of Flat GHC on the Multi-PSI. In *Fourth International Conference on Logic Programming*, pages 257–275. University of Melbourne, MIT Press, May 1987.

[Jan90] G. Janssens. *Deriving Run-time Properties of Logic Programs by means of Abstract Interpretation*. PhD thesis, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, March 1990.

[JH90] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. Technical Report PEPMA Project, SICS, Box 1263, S-164 28 KISTA, Sweden, November 1990. Forthcoming.

[JH91] Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[Kal87] L. Kale. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming*, pages 616–632. Melbourne, Australia, May 1987.

[Lus88] E. Lusk et. al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[MH89] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *1989 International Conference on Logic Programming*. MIT Press, June 1989.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie Mellon, Pittsburgh, 1989. School of Computer Science.

[SCWY90] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[Ued87a] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[Ued87b] K. Ueda. Making Exhaustive Search Programs Deterministic. *New Generation Computing*, 5(1):29–44, 1987.

[War88] D. H. D. Warren. The Andorra Model. Presented at Gigalips Project workshop. U. of Manchester, March 1988.

[War90] D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[WHD88] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699, Seattle,Washington, August 1988. MIT Press.